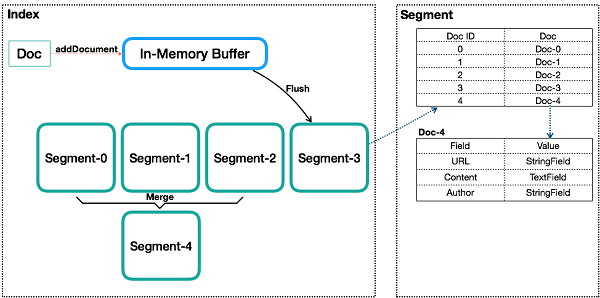
# Lucene的基本概念

<https://www.alibabacloud.com/blog/analysis-of-lucene---basic-concepts_594672?spm=a2c65.11461447.0.0.35731139NVGJOK>

1、可扩展、高性能的数据索引引擎，提供精确、高效率的数据检索算法。

2、提供全文索引和其他类型的索引操作。



3、索引（Index）在概念上类似于关系型数据库的表（Table）。但是标的结构和各种约束需要事先定义，例如字段类型、主键、外键。但是Lucene的索引不需要事先定义这些约束。

4、可以把Lucene的索引想象成一个文件夹，可以把各类文档都放到这个文件夹里面，Lucene负责根据文档内容来建立索引结构。

5、Lucene的每一个文档类似于数据库的一行数据，Lucene会为这个文档分配一个序列号，唯一标识这个文档。

6、与数据库的一行数据类似，Lucene的每个文档由许多域（Field）组成，文档的详细内容存储于这些不同的Field中。Field有不同的类型，例如StringField，TextField或是NumericDocValueField，Lucene会根据不同的Field类型采取不同的索引动作。比如针对TextField，Lucene会为对其内容进行分词操作，在分词基础上建立倒排索引，方便快速根据单词来检索文档。

7、对Field中的内容，Lucene会将其放入Analyzer中做进一步的拆分，通过分词器将其拆成一个一个的词，这些词叫做Term（项），这是Lucene构建索引以及数据检索的最小基本单位。在完成这些数据拆分后，Lucene后构造一个叫做项词典（Term Dictionary）的结构，用于进行Term快速查找。

8、在处理完新加入的文档后，会将这些文档暂时放于内存中，这时这些文档是不可搜索的。当文档数量或是大小超过一定阈值后，Lucene会把这些文档持久化到磁盘上，并且不会覆盖之前写入的数据。这些持久化的文件被称作段（Segment）。

9、每一个段都是一个独立的、可搜索的索引结构，并且不可更改。但是可被删除，通过记录被删除的文档Id来实现。检索时会依次搜索这些段，并汇总搜索结果。

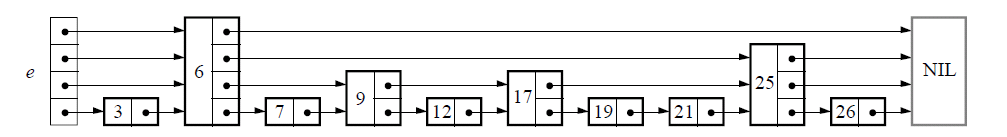
10、随着写入的数据越来越多，Lucene建立的段也会越来越多，为了提高检索效率，避免打开太多文件，Lucene会按照一定策略（段大小、包含的文档数量）合并这些段。P67

# Lucene搜索

<https://www.cnblogs.com/bonelee/p/6394451.html>

<https://www.cnblogs.com/forfuture1978/archive/2010/04/04/1704258.html>

<https://zhuanlan.zhihu.com/p/35814539> fst



# 为什么使用Lucene

对比MySQL的B+Tree索引原理，可以发现：

1）Lucene的Term index和Term Dictionary其实对应的就是MySQL的B+Tree的功能，为关键字key提供索引。Lucene的inverted index可以比MySQL的b-tree检索更快。

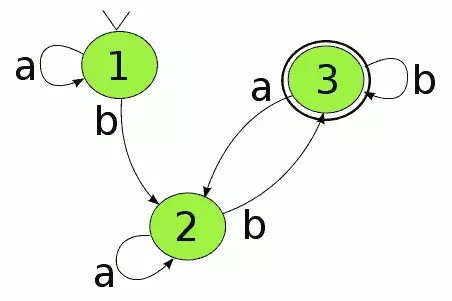
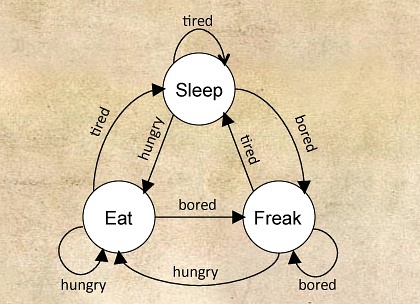
2）Term index在内存中是以FST（finite state transducers）的形式保存的，其特点是非常节省内存。所以Lucene搜索一个关键字key的速度是非常快的，而MySQL的B+Tree需要读磁盘比较。

3）Term dictionary在磁盘上是以分block的方式保存的，一个block内部利用公共前缀压缩，比如都是Ab开头的单词就可以把Ab省去。这样Term dictionary可以比B-tree更节约磁盘空间。

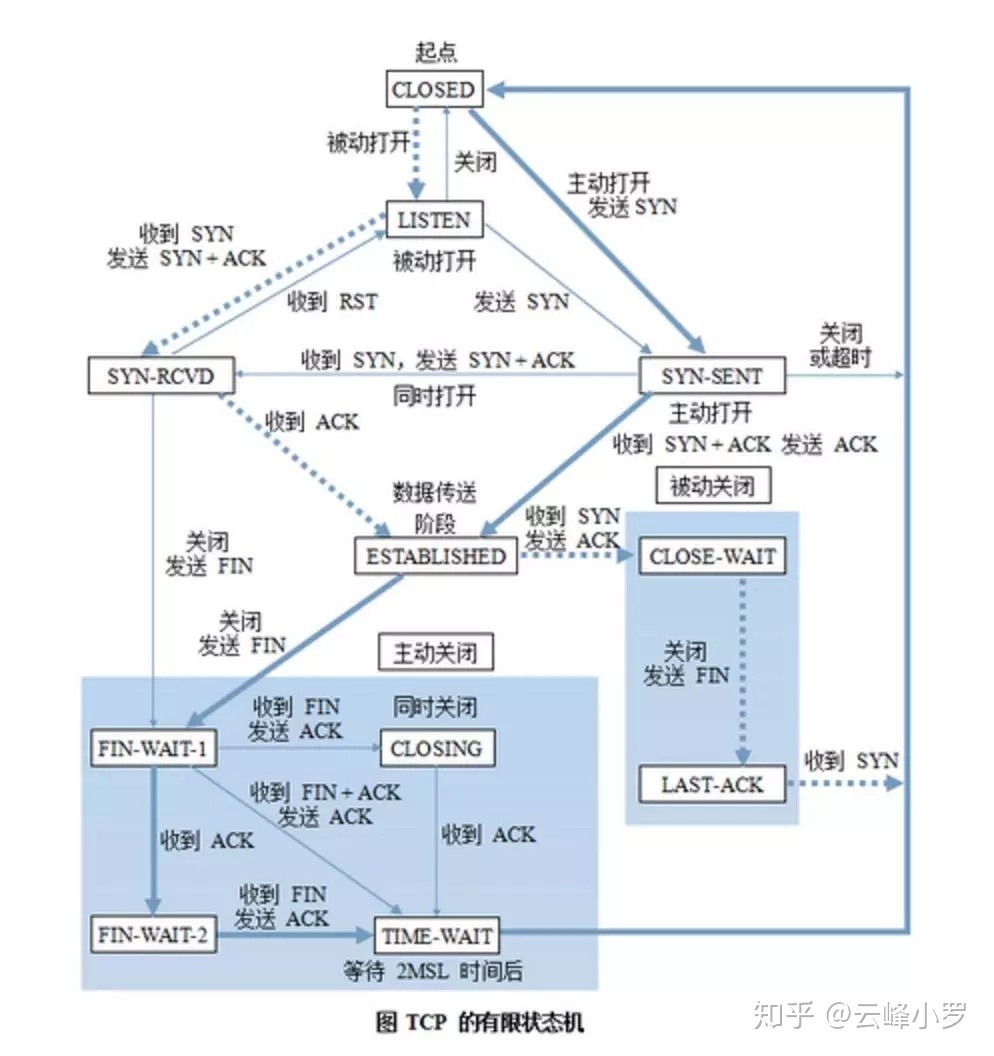
4）Lucene对不同的数据类型采用了不同的索引方式，上面分析是针对field为字符串的，比如针对int，有TrieIntField类型，针对经纬度，就可以用GeoHash编码。

5）在 Mysql中给两个字段独立建立的索引无法联合起来使用，必须对联合查询的场景建立复合索引，而Lucene可以任何AND或者OR组合使用索引进行检索。

# 有限状态机（Finite State Machine）

* 有限状态机主要是描述对象在它的生命周期内所经历的状态序列，以及如何响应来自外界的各种事件。
* 状态机可归纳为4个要素，即现态、条件、动作、次态。
* ①现态：是指当前所处的状态。  
  ②条件：当一个条件被满足，将会执行一次状态的迁移。  
  ③动作：条件满足后执行的动作。不是必须的  
  ④次态：条件满足后要迁往的新状态。
* 
* 

小白鼠的一天（3个状态，3种条件）



编程时实现相关业务逻辑时经常需要处理各种事件和状态切换，写各种switch/case 和if/else.

# 有限状态转换器（Finite State Transducers）

* FST在Lucene（>=4.x）中同义词替换功能的实现功能类似于SortedMap，很节约内存。

Finite State Transducers 简称 FST， 有限状态转换器。在自然语言处理等领域有很大应用。目前Lucene4.0在查找Term时就用到了该算法来确定此Term在字典中的位置。

      FST 可以表示成FST<Key, Value>的形式，我们可以用O（length（key））的复杂度，找到key所对应的值。除此之外，FST 还支持用Value来查找key以及查找Value最优的key等功能。

* 使用lucene进行查询不可避免需要根据给定的term找到该term所对应的倒排文档id列表等信息。
* Lucene的倒排索引：



* 可以采用HashMap， TRIE， Binary Search Tree， Tenary Search Tree等各种数据结构来实现倒排索引
* FST优势：

1）空间占用小。通过对词典中单词前缀和后缀的重复利用，压缩了存储空间；

2）查询速度快。O(len(str))的查询时间复杂度。

# FSA示例

<https://blog.csdn.net/AAA821/article/details/82014792>

<http://examples.mikemccandless.com/fst.py>

<https://www.shenyanchao.cn/blog/2018/12/04/lucene-fst/>

* 有限状态接收器finite state acceptor
* 确定：意味着指定任何一个状态，只可能最多有一个转移可以访问到。

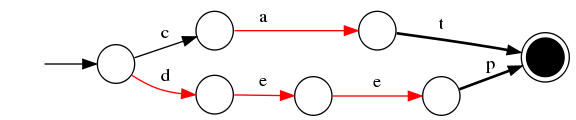
无环： 不可能重复遍历同一个状态

接收机：有限状态机只“接受”特定的输入序列，并终止于final状态。

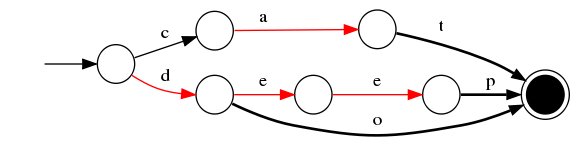
* 一种压缩，让每个节点都不会放重复值，节省了大量的空间
* 对" cat"、 " deep"、 " do"、 " dog" 、" dogs"这5个单词进行插入构建FSA。假设需要存储这几个单词，有哪些方式，数据结构。
* 插入cat，每个字母形成一条边，其中t边指向终点。

https://images0.cnblogs.com/blog/522490/201411/242224195125261.png

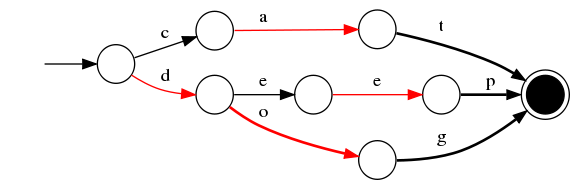
* 插入“deep”,与前一个单词“cat”进行最大前缀匹配，发现没有匹配则直接插入，P边指向终点。



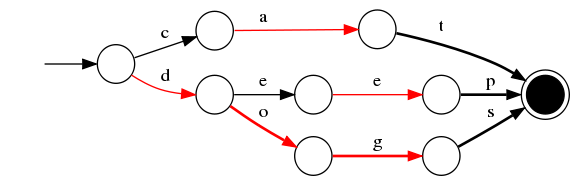
* 插入“do”,与前一个单词“deep”进行最大前缀匹配，发现是d，则在d边后增加新边o，o边指向终点。



* 插入“dog”,与前一个单词“do”进行最大前缀匹配，发现是do，则在o边后增加新边g，g边指向终点。



* 插入“dogs”,与前一个单词“dog”进行最大前缀匹配，发现是dog，则在g后增加新边s，s边指向终点。

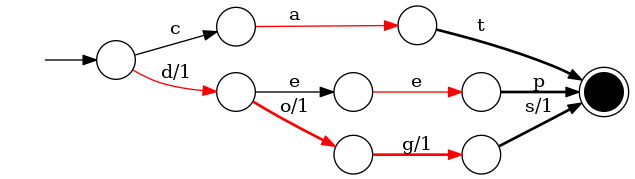


* 有向无环图。利用该结构可以很方便的进行查询，如给定一个term “dog”，我们可以通过上述结构很方便的查询存不存在，甚至我们在构建过程中可以将单词与某一数字、单词进行关联，从而实现key-value的映射。也就是它这种压缩，就是让每个节点都不会放重复值，节省了大量的空间
* FST以字节的方式存储所有的term，这种压缩方式可以有效的缩减存储空间(优点)，使得term足以放进内存，但这种方式也会导致查找时需要更多的CPU资源

# FST的构建

需要建立映射关系：

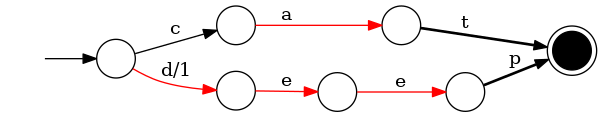
* term: cat, deep, do, dog, dogs
* docId: 0, 1, 2, 3, 4



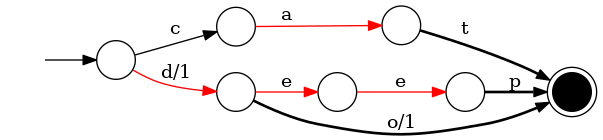
第一步，建立cat-0：

imgs/animal_all_01.png

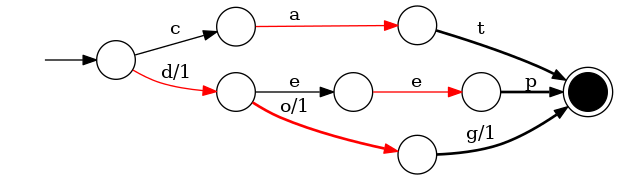
第二步，建立deep-1：



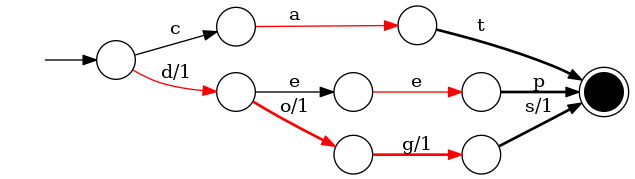
第三步，建立do-2：



第四步，建立dog-3：

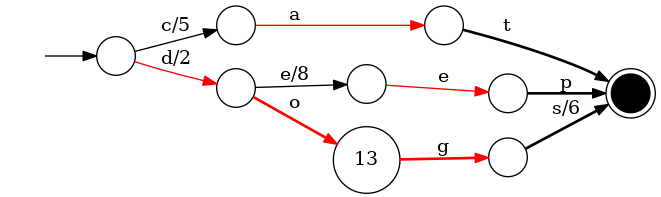


第五步，建立dogs-4：

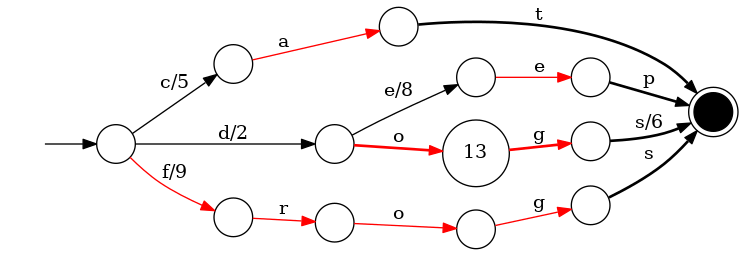


需要建立映射关系：

* term: cat, deep, do, dog, dogs
* docId: 5, 10, 15, 2, 8



* term: cat, deep, do, dog, dogs,frogs
* docId: 5, 10, 15, 2, 8, 9



# IndexWriter，写、更新、删除操作基本原理

<https://blog.51cto.com/sbp810050504/1440510>

<https://www.jianshu.com/p/4519371b4607>

<https://zhuanlan.zhihu.com/p/35795070>

<https://www.alibabacloud.com/blog/lucene-indexwriter-an-in-depth-introduction_594673>

1、Lucene索引一个文档分为写内存与写磁盘两个阶段。

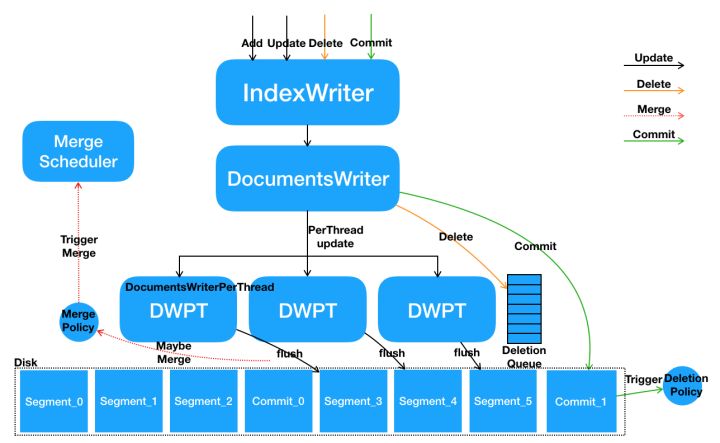
2、写内存阶段，通过索引链（IndexChain），将文档分解，并将相关信息存入内存。

3、当满足一定条件后（文档占用内存大小或是文档数量超出阈值），通过索引链（IndexChain），把内存数据写入磁盘。

4、索引文档的过程就是分解文档的过程，通过分解文档，得到词典、倒排表等信息。索引链（IndexChain）就是这样一个分解文档的架构，由一组对象组成。

5、索引链分为主要分为两组，分别处理文档的正向信息（通过id找文档）与反向信息（通过项找文档）。正向信息包括文档、域、项的查找关系，通过文档id可以找出文档的域，进而可以得到文档的内容。反向信息通过域值（Term）查找对应的文档。

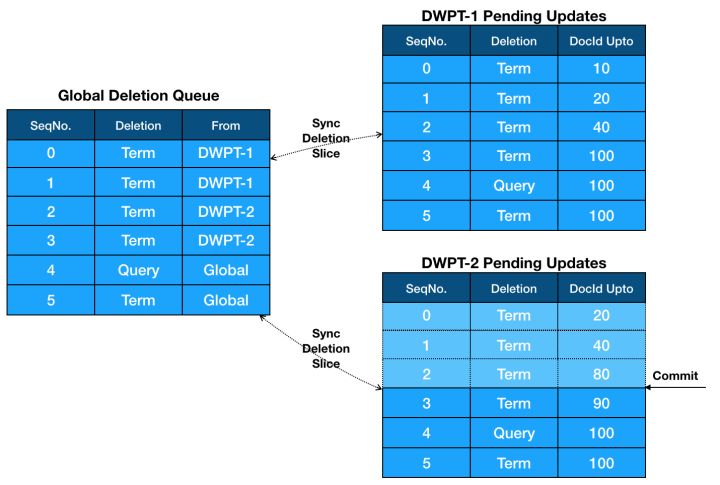
6、Lucene支持多线程同时构建索引，每个线程都持有一个索引链，互不干扰。



7、IndexWriter提供的核心接口都是线程安全的，并且内部做了特殊的并发优化来优化多线程写入的性能。IndexWriter内部为每个线程都会单独开辟一个空间来写入，这块空间由DocumentsWriterPerThread来控制。

8、IndexWriter的具体索引操作会交由DocumentsWriter执行。在处理具体请求前，DocumentsWriter会通过DocumentsWriterPerThreadPool获取一个DocumentsWriterPerThread，每个 DocumentsWriterPerThread具有一块独立的内存空间，在这块独立的内存空间内进行分词、构建索引等数据处理操作。在触发Flush动作后，DocumentsWriterPerThread管理的内存空间内的数据会被存入磁盘，形成不同的segment文件。

9、Lucene的文档更新操作与数据库不同，是先查询、再删除，最后插入新文档。在DocumentsWriter的执行流程为，从DocumentsWriterPerThreadPool取出一个DocumentsWriterPerThread，在DocumentsWriterPerThread执行delete，之后再执行add操作。



10、在Delete路径上关键的数据结构就是Deletion queue，在IndexWriter内部会有一个全局的Deletion Queue，称为Global Deletion Queue，而在每个DWPT内部，还会有一个独立的Deletion Queue，称为Pending Updates。DWPT Pending Updates会与Global Deletion Queue进行双向同步，因为文档删除是全局范围的，不应该只发生在DWPT范围内。

update只能进行by term的文档删除，而delete除了by term，还支持by query。

update的删除会先作用于DWPT内部，后作用于Global，再由Global同步到其他DWPT。

delete的删除会作用在Global级别，后异步同步到DWPT级别。

update的删除操作会先发生在DWPT内部，并且是和add同时发生，所以能够保证该DWPT内部的delete和add的***原子性***，即保证在add之前的所有符合条件的文档一定被删除。

11、DWPT Pending Updates里的删除操作什么时候会真正作用于数据呢？在Lucene Segment内部，数据实际上并不会被真正删除。Segment中有一个特殊的文件叫live docs，内部是一个位图的数据结构，记录了这个Segment内部哪些DocId是存活的，哪些DocId是被删除的。所以删除的过程就是构建live docs标记位图的过程，数据实际上不会被真正删除，只是在live docs里会被标记删除。Term删除和Query删除会在不同阶段构建live docs，Term删除要求先根据Term查询出它关联的所有doc，所以很明显这个会发生在倒排索引构建时。而Query删除要求执行一次完整的查询后才能拿到其对应的docId，所以会发生在segment被flush完成后，基于flush后的索引文件构建IndexReader后执行搜索才能完成。

12、live docs只影响倒排，所以在live docs里被标记删除的文档没有办法通过倒排索引检索出，但是还能够通过doc id查询到store fields。当然文档数据最终是会被真正物理删除，这个过程会发生在merge时。

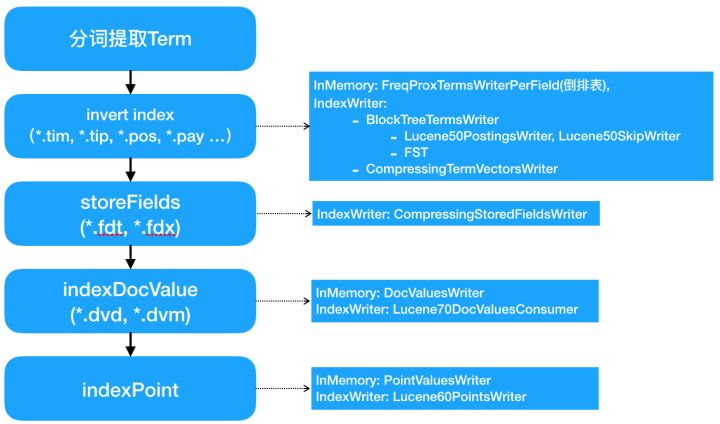
13、flush是将DWPT内In-memory buffer里的数据持久化到文件的过程，flush会在每次新增文档后由FlushPolicy判定自动触发，也可以通过IndexWriter的flush接口手动触发。 每个DWPT会flush成一个segment文件，flush完成后这个segment文件是不可被搜索的，只有在commit之后，所有commit之前flush的文件才可被搜索。

14、commit时会触发数据的一次强制flush，commit完成后再此之前flush的数据才可被搜索。commit动作会触发生成一个commit point，commit point是一个文件。Commit point会由IndexDeletionPolicy管理，lucene默认配置的策略只会保留last commit point，当然lucene提供其他多种不同的策略供选择。

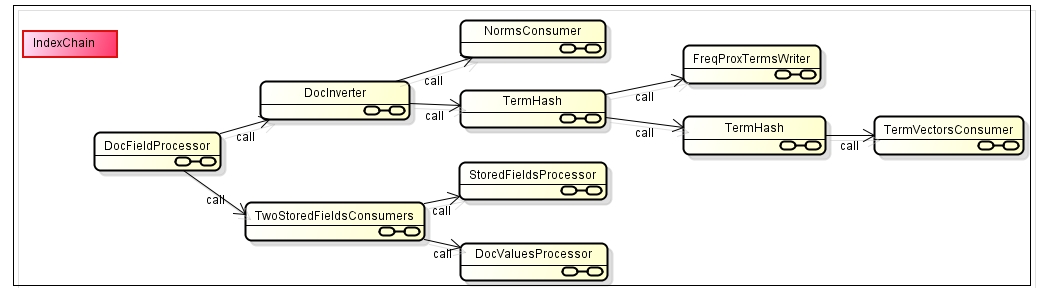
15、merge是对segment文件合并的动作，合并的好处是能够提高查询的效率以及回收一些被删除的文档。Merge会在segment文件flush时触发MergePolicy来判定自动触发，也可通过IndexWriter进行一次force merge。

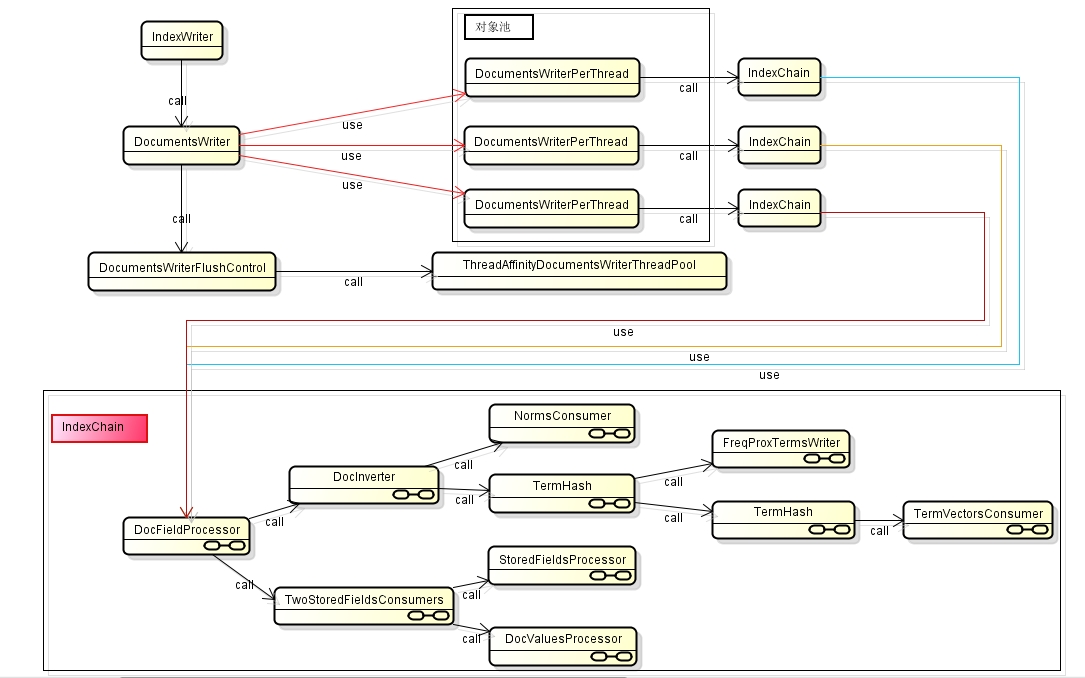
# IndexWriter，构建索引过程，索引链

1、Lucene内部索引构建最关键的概念是IndexingChain。Lucene提供了各种不同类型的索引类型，例如倒排、正排（列存）、StoreField、DocValues等。每个不同的索引类型对应不同的索引算法、数据结构以及文件存储，有些是列级别的，有些是文档级别的。所以一个文档写入后，需要被这么多种不同索引处理，有些索引会共享memory-buffer，有些则是完全独立的。基于这个架构，理论上Lucene是提供了扩展其他类型索引的可能性。



2、<https://blog.51cto.com/sbp810050504/1440510>





这种设计导致IndexChain只是一个骨架. 每个索引线程(dwpt)其实是从DocumentsWriterPerThreadPool里面获得空闲的DocumentsWriterPerThread对象

# IndexWriter，构建索引过程，内存管理（倒排索引）

<https://blog.51cto.com/sbp810050504/1440572>

1、索引过程中，通过ByteBlockPool,IntBlockPool, ParallelPostingsArray三个类来协调配合存储数据。

2、ByteBlockPool与IntBlockPool是索引链级的，每个线程都会分别持有。

文档Id、词频以及词（Term）出现在文档中的位置存于ByteBlockPool中。

3、ParallelPostingsArray的作用域是Field，跨越多个索引链。

4、ByteBlockPool与IntBlockPool通过一个一个的Slice来分配内存，最后将这些Slice连接起来，以链表的形式管理内存。一共有9种不同Level的Slice，每个Level又分为10种不同大小的Slice，最小5字节，最大200字节。

# IndexWriter，构建索引过程，内存管理（正向信息）

1、正向信息存储文档Id到文档内容的映射。

2、将文档域值（Field）依次放入一个可变大小的byte数组中（GrowableByteArrayDataOutput），并将每个Field的偏移量存于另一个int类型数组中。

# IndexWriter，构建索引过程，内存管理（数值存储）

1、Lucene为了改善内存使用效率，采用了位压缩技术（bit-packing）。

2、将Java的整数类型byte（8）、short（16）、int（32）、long（64）这四种类型，扩展到了1-64位共64种类型。

3、存储数字4，100，3位，Java中最小使用一个ubyte存储，8位，浪费了63%的空间。

4、对于位压缩，Lucene有不同的实现。

Direct8 –> byte[], Direct16 -> short[], Direct32 -> int[], Direct64 -> long[]

用于存储从流中读取的数据，比如小于1字节的数用Direct8存储，大于1字节但小于等于2字节的数用Direct16存储。高效率，但仍有空间浪费。

Packed64，将数据连续存储于64位大小的块中（long[] blocks），若一个块中的空间不够存储一个数字，则将该数字拆分存于相邻的两个块中。

Packed64SingleBlock，与Packed64不同之处在于数字不用跨越块，性能比Packed64高，但存在空间浪费。

Packed8ThreeBlock，使用3个Byte（24位）存储一个数。

Packed16ThreeBlock，使用3个Short（48位）存储一个数。

以Packed64为例，存储三个30位大小的数字，跨越两个Block

# IndexWriter，构建索引过程，内存管理（LZ4压缩算法）

<https://blog.51cto.com/sbp810050504/1532152>

<http://fastcompression.blogspot.com/p/lz4.html>

1、LZ4算法又称为Realtime Compression Algorithm，压缩与解压速度非常快，在一块2.7GHz的i5处理器上，单线程的压缩/解压速度可达，422/1820MB/s。

2、LZ4算法注重压缩和解压速度，压缩比并不是第一位

3、安卓和苹果操作系统中，内存压缩技术就使用的是lz4算法

4、输入：abcde\_bcdefgh\_abcdefghxxxxxxx

输出：abcde\_(5,4)fgh\_(14,5)fghxxxxxxx

(5,4) 代表向前5个byte，匹配到的内容长度有4，即"bcde"是一个重复

输入：fghabcde\_bcdefgh\_abcdefghxxxxxxx

输出：fghabcde\_(5,4)(13,3)\_(14,5)fghxxxxxxx

# IndexWriter，持久化索引过程描述

1、生成的第一个文件是write.lock，临时文件，索引创建结束后会被删除。保证在任一时刻，只有一个IndexWriter在修改索引数据。

2、对每个文档，解析完其域（Fields）中的内容后，（分词操作），得到许多项（Term），将解析完成后的域值存储到文件中（fdt、fdx）。

3、fdt文件存储的是文档的域（Fields）数据，每个文档依次顺序存储，fdx存储fdt文件的域索引结构。通过fdx的索引，可以实现对文档以及域的快速随机访问。

4、fnm文件存储索引的Field结构信息，不包括值。

5、存储索引的倒排表信息。

6、存储每个项（Term）在对应文档域（Field）中的位置信息。

7、存储索引的加权信息，包含三块，文档加权、域加权和域值长度加权信息（1/sqrt(length)），这些数值会影响文档搜索时的评估分值。

8、生成索引的segment\_N文件，此文件存储了索引的段信息，即索引包含了多少段，每个段的段名、段的大小、段包含Document数目等信息。N表示索引的代数（generation），每次改变索引都会改变这个数字。

9、如果使用复合文件（CompoundFile），会将以上文件打包成cfs与cfe文件，减少同时打开的文件数量。